Project 6: Virtual Memory I (4%)  

ENEE 447: Operating Systems — Spring 2016  
Assigned: Monday, Mar 21; Due: Friday, Apr 1

Purpose
In this project you will figure out how to turn on the ARM’s virtual memory system. Virtual memory underlies many of computing’s most important facilities, including process protection, shared memory, multitasking, the kernel’s privileged mode, the familiar virtual-machine programming model, and more. It is essential to most operating systems, especially general-purpose operating systems. Your implementation will be very simple but will have all of the essentials, including shared pages (two different virtual pages mapping to the same physical page), different mapping characteristics for different pages, etc. Once you get this working, it should become obvious how to extend the facility to a full system, as we will in a future project.

Virtual Memory and the ARM/Raspberry Pi
Address translation is the mechanism through which the operating system provides virtual address spaces to user-level applications. The operating system maintains a set of mappings that translate references within the per-process virtual spaces to the system’s physical space. Addresses are usually mapped at a page granularity—typically several kilobytes. The mappings are organized in a page table, and for performance reasons most hardware systems provide a translation lookaside buffer (TLB) that caches those PTEs (page-table entries; i.e. mappings) that have been needed recently. When a process performs a load or store to a virtual address, the hardware translates this to a physical address using the mapping information in the TLB. If the mapping is not found in the TLB, it must be retrieved from the page table and loaded into the TLB before processing can continue. The ARM has a TLB, and its hardware can automatically walk the page tables and load the TLB with the required information, when it find it in the page table.

The table looks like this:

![Figure B3-3 General view of address translation using Short-descriptor format translation tables](image)

Note that there is one page in the first-level table and potentially thousands of pages making up the second-level table. However, if the PTE indicates that it maps a large area, like a 1MB “section” or a 16MB “supersection,” then there need be no second-level table at all. The PTE format looks like this:
Short-descriptor translation table first-level descriptor formats

Each entry in the first-level table describes the mapping of the associated 1MB memory virtual address (MVA) range. Figure B3-4 shows the possible first-level descriptor formats.

Inclusion of the PXN attribute in the short-descriptor translation table formats is:

- **OPTIONAL** in an implementation that does not include the Large Physical Address Extension.
- **required** in an implementation includes the Large Physical Address Extension.

Descriptor bits [1:0] identify the descriptor type. On an implementation that supports the PXN attribute, for the Section and Supersection entries, bit [0] also defines the PXN value. The encoding of these bits is:

- 0b00, **Invalid**: The associated virtual address (VA) is unmapped, and any attempt to access it generates a translation fault.
- 0b01, **Page table**: The descriptor gives the address of a second-level translation table, that specifies the mapping of the associated 1MByte VA range.
- 0b10, **Reserved, UNK/SBZP**: Reserved, when Large Physical Address Extension not implemented.
- 0b11, **Reserved, SBZP**: Reserved, when Large Physical Address Extension implemented.
- 0b10, **Reserved, UNK/SBZP**: Reserved, when Large Physical Address Extension not implemented.
- 0b11, **Reserved, SBZP**: Reserved, when Large Physical Address Extension implemented.

Putting 0b10 in the bottom two bits indicates that the PTE is for a 1MB section.

**Your First-Ever VM Implementation**

We will implement the simplest of facilities: a single level page table (just an array, really) of page-table entries (PTEs) indexed by the virtual page number. Our page sizes will be the 1MB sections, so the page table need only hold 4K entries to map the entire 4GB space. Using large pages allows the table to be relatively small: 16KB per page table.

The code on core1 is the test subject: this is the code that will experience virtual memory. It runs a loop in SYS mode that just flashes the LEDs, but it flashes the red LED using GPIO addresses and flashes the green LED using addresses in “virtual page 2” … so, at least when you first get your code and start experimenting with it, the green LED will not flash. This is because the “user-level” code (not really user-level because it runs in SYS mode) calls two different functions:

- **flash_led**, which uses the following addresses:
  - #define GPFSEL3 0x3F20000C
  - #define GPFSEL4 0x3F200010
  - #define GPSET1 0x3F200020
  - #define GPCLR1 0x3F20002C
- **flash_led_diffio**, which uses different I/O addresses:
  - #define V_GPFSEL3 0x0020000C
  - #define V_GPFSEL4 0x00200010
  - #define V_GPSET1 0x00200020
  - #define V_GPCLR1 0x0020002C

© Copyright 2016 Bruce Jacob, All Rights Reserved
Note that, if a page size is 1MB, then the bottom 20 bits are page-offset bits, and the topmost 12 bits create the virtual page number. Thus an address looks like the following in hex:

0xVVVOOOOO

Where the “V” bits make up the virtual page number, and the “O” bits make up the page offset.

In the Raspberry Pi 2, the GPIO registers (including the ones corresponding to the LEDs) are memory-mapped to the 0x3F2xxxxx addresses, so when the `flash_led_diffio()` function uses the V_GPx addresses which are in the 0x002xxxxx range, no GPIOs are affected, and thus the LED does nothing. The function blindly sends out data values into a garbage area of memory.

The code on `core0` tells `core1` when to initiate virtual memory, by interrupting `core1` after a few seconds. Five seconds after system start-up, `core0` interrupts `core1`, and in the IRQ interrupt handler, `core1` calls the function `enable_vm()`, which turns everything on. This will be transparent to the blinking code running on `core1` — it should transitions from one moment, during which only the red LED is blinking, to the next moment, where either the green LED is blinking, or both red and green LEDs are blinking, depending on how you choose to map the IO addresses.

The kernel code on `core0` at the outset initializes the user page tables to 0s … in other words, all PTEs are invalid at startup. Thus, the `enable_vm()` routine needs only to set a handful of PTEs and then turn the correct switches to get the TLB operational. There are only four distinct pages being used by your code at the moment the `enable_vm()` function is called:

- 0x3F2xxxxx — GPIO addresses
- 0x002xxxxx — V_GPIO addresses (virtual addresses that, until the virtual memory system is enabled, point out into the wilderness and do nothing)
- 0x400xxxxx — timer/clock device-register addresses
- 0x000xxxxx — where nearly all your code and data lies

You will want to create a mapping for each, as follows:

- 0x3F2xxxxx → both 0x3F2xxxxx and somewhere else, for example 0x002xxxxx (First have the GPIO addresses map to themselves, but also point them to 0x002xxxxx, for symmetry’s sake: the first five seconds only the red LED blinks, and then only the green one will)
- 0x002xxxxx → 0x3F2xxxxx
- 0x400xxxxx → 0x400xxxxx
- 0x000xxxxx → 0x000xxxxx

Note that we are cheating a bit here, because what we are doing is running an application in bare-metal mode, in which it is executing out of physical memory using physical addresses, and we are using virtual memory to translate just one select range of virtual addresses, in a sort of contrived way. Don’t worry, we will get to the real thing by the end of the semester.

**ARM Documentation**

You will find the *ARM Architecture Reference Manual* to be invaluable. I will point out some of the most important pages, but you need to explore this document yourself, because the information that you need is spread out all over the document. This is one of those (perhaps many) instances in which you curse ARM, because they really are a misnomer: ARM stands for Acorn RISC Machines, and RISC means Reduced Instruction-Set Computer … any computer architecture that requires tens of thousands of pages of documentation cannot, in any way, shape, or form, be considered “reduced” …
Short-descriptor translation table first-level descriptor formats

Each entry in the first-level table describes the mapping of the associated 1MB MVA range. Figure B3-4 shows the possible first-level descriptor formats.

Inclusion of the PXN attribute in the Short-descriptor translation table formats is:
- **OPTIONAL** in an implementation that does not include the Large Physical Address Extension
- **required** in an implementation includes the Large Physical Address Extension.

Descriptor bits[1:0] identify the descriptor type. On an implementation that supports the PXN attribute, for the Section and Supersection entries, bit[0] also defines the PXN value. The encoding of these bits is:

- **0b00**, Invalid
  The associated VA is unmapped, and any attempt to access it generates a Translation fault.
  Software can use bits[31:2] of the descriptor for its own purposes, because the hardware ignores these bits.

- **0b01**, Page table
  The descriptor gives the address of a second-level translation table, that specifies the mapping of the associated 1MByte VA range.

- **0b10**, Domain
- **0b11**, Reserved, when Large Physical Address Extension not implemented

† If the implementation does not support the PXN attribute this bit is SBZ.
‡ If the implementation does not support the PXN attribute these bits must be 0.

This is a picture of what the format of the PTE is ... each of the bits has meaning, and the following handful of pages in the Architectural Reference Manual go into detail (and some are described much later in the document). Pay close attention to the bits involved in how the memory behaves (e.g., caching), because some of the settings are specifically for I/O addresses.

Note: in this project we are re-routing I/O addresses through the TLB. I suspect this is unusual, except for hypervisor/guest-operating-system configurations, because the OS often runs in physical mode and is the only one allowed to touch the devices. We will have a split-personality OS by the end of class.
TTBCR format when using the Short-descriptor translation table format

In an implementation that includes the Security Extensions and is using the Short-descriptor translation table format, the TTBCR bit assignments are:

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| 31  | 30  | 23  | 22  | 21  | 20  | 19  | 18  |
|      |     |     |     |     |     | EAE |     |
|      |     |     |     |     |     |     | 0   |
|      |     |     |     |     |     |     | N   |
```

† Reserved, UNK/SBZP, if the implementation does not include the Large Physical Address Extension.

In an implementation that does not include the Security Extensions, and is using the Short-descriptor translation table format, the TTBCR bit assignments are:

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| 31  | 30  | 23  | 22  | 21  | 20  | 19  | 18  |
|      |     |     |     |     |     |     |     |
|      |     |     |     |     |     | EAE |     |
|      |     |     |     |     |     |     | 0   |
|      |     |     |     |     |     |     | N   |
```

† Reserved, UNK/SBZP, if the implementation does not include the Large Physical Address Extension.

EAE, bit[31], if implementation includes the Large Physical Address Extension

Extended Address Enable. The meanings of the possible values of this bit are:

- 0 Use the 32-bit translation system, with the Short-descriptor translation table format. In this case, the format of the TTBCR is as described in this section.
- 1 Use the 40-bit translation system, with the Long-descriptor translation table format. In this case, the format of the TTBCR is as described in TTBCR format when using the Long-descriptor translation table format on page B4-1726.

This bit resets to 0, in both the Secure and the Non-secure copies of the TTBCR.

Bit[31], if implementation does not include the Large Physical Address Extension

Reserved, UNK/SBZP.

Bits[30:6, 3] Reserved, UNK/SBZP.

PD1, bit[5], in an implementation that includes the Security Extensions

Translation table walk disable for translations using TTBR1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1. The encoding of this bit is:

- 0 Perform translation table walks using TTBR1.
- 1 A TLB miss on an address that is translated using TTBR1 generates a Translation fault. No translation table walk is performed.

PD0, bit[4], in an implementation that includes the Security Extensions

Translation table walk disable for translations using TTBR0. This bit controls whether a translation table walk is performed on a TLB miss for an address that is translated using TTBR0. The meanings of the possible values of this bit are equivalent to those for the PD1 bit.

Bits[5:4], in an implementation that does not include the Security Extensions

Reserved, UNK/SBZP.

This is the TTBCR, the register that determines how big the page size is via the N bits.
B4.1.154 TTBR0, Translation Table Base Register 0, VMSA

The TTBR0 characteristics are:

**Purpose**
TTBR0 holds the base address of translation table 0, and information about the memory it occupies. This is one of the translation tables for the stage 1 translation of memory accesses from modes other than Hyp mode.

This register is part of the Virtual memory control registers functional group.

**Usage constraints**
Only accessible from PL1 or higher.

Used in conjunction with the TTBCR. When the 64-bit TTBR0 format is used, cacheability and shareability information is held in the TTBCR, not in TTBR0.

**Configurations**
The Multiprocessing Extensions change the TTBR0 32-bit register format.

The Large Physical Address Extension extends TTBR0 to a 64-bit register. In an implementation that includes the Large Physical Address Extension, TTBCR.EAE determines which TTBR0 format is used:

- EAE=0 32-bit format is used. TTBR0[63:32] are ignored.
- EAE=1 64-bit format is used.

If the implementation includes the Security Extensions, this register:
- is Banked
- has write access to the Secure copy of the register disabled when the CP15DISABLE signal is asserted HIGH.

**Attributes**
A 32-bit or 64-bit RW register with a reset value that depends on the register implementation. For more information see the register bit descriptions. See also Reset behavior of CP14 and CP15 registers on page B3-1450.

Table B3-45 on page B3-1493 shows the encodings of all of the registers in the Virtual memory control registers functional group.

The following subsections describe the TTBR0 formats:
- 32-bit TTBR0 format
- 64-bit TTBR0 and TTBR1 format on page B4-1731.

See TTBCR, Translation Table Base Control Register, VMSA on page B4-1724 for more information about using this register.

**Note**
See TTBCR, Translation Table Base Control Register, VMSA on page B4-1724 for a summary of the registers that define the translation tables for other address translations.

### 32-bit TTBR0 format

In an implementation that does not include the Multiprocessing Extensions, the 32-bit TTBR0 bit assignments are:

<table>
<thead>
<tr>
<th>31</th>
<th>30...20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>28...18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>27</td>
<td>26...16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>26</td>
<td>25...15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>25</td>
<td>24...14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>24</td>
<td>23...13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>23</td>
<td>22...12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>21...11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>20...10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

This is the TTBR0 register; there is also a TTBR1 register. These contain the address of the page table for the currently executing process. When you context switch to another running *process* (which has a different address space, as opposed to switching to another *thread*, which doesn’t), you need to give the hardware the pointer to the new process’s address space.
Figure B3-3 gives a general view of address translation when using the Short-descriptor translation table format.

The first-level entries point to second-level entries, which point to the actual page data. When the first-level entries identify themselves as “sections” they instead point directly to page data.
In the selected TTBR, the following bits define the memory region attributes for the translation table walk:

- the RGN, S, and C bits, in an implementation that does not include the Multiprocessing Extensions
- the RGN, S, and IRGN[1:0] bits, in an implementation that includes the Multiprocessing Extensions.

For more information, see TTBCR, Translation Table Base Control Register, VMSA on page B4-1724, TTBR0, Translation Table Base Register 0, VMSA on page B4-1729 and TTBR1, Translation Table Base Register 1, VMSA on page B4-1733.

Translation table walks, when using the Short-descriptor translation table format, describes the translation.

B3.5.5 Translation table walks, when using the Short-descriptor translation table format

When using the Short-descriptor translation table format, and a memory access requires a translation table walk:

- a section-mapped access only requires a read of the first-level translation table
- a page-mapped access also requires a read of the second-level translation table.

Reading a first-level translation table describes how either TTBR1 or TTBR0 is used, with the accessed VA, to determine the address of the first-level descriptor.

Reading a first-level translation table shows the output address as A[39:0]:

- On an implementation that includes the Virtualization Extensions, for a Non-secure PL1&0 stage 1 translation, this is the IPA of the required descriptor. A Non-secure PL1&0 stage 2 translation of this address is performed to obtain the PA of the descriptor.
- Otherwise, this address is the PA of the required descriptor.

The full translation flow for Sections, Supersections, Small pages and Large pages on page B3-1332 then shows the complete translation flow for each valid memory access.

Reading a first-level translation table

When performing a fetch based on TTBR0:

- the address bits taken from TTBR0 vary between bit[31:14] and bit[31:7]
- the address bits taken from the VA, that is the input address for the translation, vary between bit[31:20] and bit[24:20].

The width of the TTBR0 and VA fields depend on the value of TTBCR.N, as Figure B3-7 on page B3-1332 shows.

This indicates how the system behaves wrt multiple multiple simultaneous mappings (e.g. split between two different guest operating systems). One is mapped through the TTBR0 page table, and the other is mapped through the TTBR1 page table, and the amount of memory assigned to each is variable.
B3 Virtual Memory System Architecture (VMSA)

B3.5 Short-descriptor translation table format

B3.5.3 Control of Secure or Non-secure memory access, Short-descriptor format

Access to the Secure or Non-secure physical address map on page B3-1321 describes how the NS bit in the translation table entries:

- for accesses from Secure state, determines whether the access is to Secure or Non-secure memory
- is ignored by accesses from Non-secure state.

In the Short-descriptor translation table format, the NS bit is defined only in the first-level translation tables. This means that, in a first-level Page table descriptor, the NS bit defines the physical address space, Secure or Non-secure, for all of the Large pages and Small pages of memory described by that table.

The NS bit of a first-level Page table descriptor has no effect on the physical address space in which that translation table is held. As stated in Secure and Non-secure address spaces on page B3-1323, the physical address of that translation table is in:

- the Secure address space if the translation table walk is in Secure state
- the Non-secure address space if the translation table walk is in Non-secure state.

This means the granularity of the Secure and Non-secure memory spaces is 1MB. However, in these memory spaces, table entries can define physical memory regions with a granularity of 4KB.

B3.5.4 Selecting between TTBR0 and TTBR1, Short-descriptor translation table format

As described in Determining the translation table base address on page B3-1320, two sets of translation tables can be defined for each of the PL1&0 stage 1 translations, and TTBR0 and TTBR1 hold the base addresses for the two sets of tables. When using the Short-descriptor translation table format, the value of TTBCR.N indicates the number of most significant bits of the input VA that determine whether TTBR0 or TTBR1 holds the required translation table base address, as follows:

- If N = 0 then use TTBR0. Setting TTBCR.N to zero disables use of a second set of translation tables.
- If N > 0 then:
  - if bits [31:32-N] of the input VA are all zero then use TTBR0
  - otherwise use TTBR1.

Table B3-1 shows how the value of N determines the lowest address translated using TTBR1, and the size of the first-level translation table addressed by TTBR0.

<table>
<thead>
<tr>
<th>TTBCR.N</th>
<th>First address translated with TTBR1</th>
<th>TTBR0 table size</th>
<th>Index range</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000</td>
<td>TTBR1 not used</td>
<td>16KB</td>
<td>VA[31:20]</td>
</tr>
<tr>
<td>0b001</td>
<td>0x80000000</td>
<td>8KB</td>
<td>VA[30:20]</td>
</tr>
<tr>
<td>0b010</td>
<td>0x40000000</td>
<td>4KB</td>
<td>VA[29:20]</td>
</tr>
<tr>
<td>0b011</td>
<td>0x20000000</td>
<td>2KB</td>
<td>VA[28:20]</td>
</tr>
<tr>
<td>0b100</td>
<td>0x10000000</td>
<td>1KB</td>
<td>VA[27:20]</td>
</tr>
<tr>
<td>0b101</td>
<td>0x80000000</td>
<td>512 bytes</td>
<td>VA[26:20]</td>
</tr>
<tr>
<td>0b110</td>
<td>0x40000000</td>
<td>256 bytes</td>
<td>VA[25:20]</td>
</tr>
<tr>
<td>0b111</td>
<td>0x20000000</td>
<td>128 bytes</td>
<td>VA[24:20]</td>
</tr>
</tbody>
</table>

Whenever TTBCR.N is nonzero, the size of the translation table addressed by TTBR1 is 16KB.

Figure B3-6 on page B3-1331 shows how the value of TTBCR.N controls the boundary between VAs that are translated using TTBR0, and VAs that are translated using TTBR1.

Table B3-1 Effect of TTBCR.N on address translation, Short-descriptor format

Whenever TTBCR.N is nonzero, the size of the translation table addressed by TTBR1 is 16KB.

These are the values that indicate how much space goes to the TTBR0 address space, and how much goes to the TTBR1 address space.
Banked system control registers

In an implementation that includes the Security Extensions, some system control registers are Banked. Banked system control registers have two copies, one Secure and one Non-secure. The SCR NS bit selects the Secure or Non-secure copy of the register. Table B3-33 shows which CP15 registers are Banked in this way, and the permitted access to each register. No CP14 registers are Banked.

### Table B3-33 Banked CP15 registers

<table>
<thead>
<tr>
<th>CRn</th>
<th>Banked register</th>
<th>Permitted accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0</td>
<td>CSSEL, Cache Size Selection Register</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td>c1</td>
<td>SCtrl, System Control Register</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td></td>
<td>ACTLR, Auxiliary Control Register</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td>c2</td>
<td>TTBR0, Translation Table Base 0</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td></td>
<td>TTBR1, Translation Table Base 1</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td></td>
<td>TTBR, Translation Table Base Control</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td>c3</td>
<td>DACR, Domain Access Control Register</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td>c4</td>
<td>DFSR, Data Fault Status Register</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td>c5</td>
<td>IFSR, Instruction Fault Status Register</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td></td>
<td>ADFSR, Auxiliary Data Fault Status Register</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td></td>
<td>AIFS, Auxiliary Instruction Fault Status Register</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td>c6</td>
<td>DFAR, Data Fault Address Register</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td></td>
<td>IFAR, Instruction Fault Address Register</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td>c7</td>
<td>PAR, Physical Address Register</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td>c8</td>
<td>PRRR, Primary Region Remap Register</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td>c9</td>
<td>NMRR, Normal Memory Remap Register</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td>c10</td>
<td>VBAR, Vector Base Address Register</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td>c11</td>
<td>FCSEIDR, FCSE PID Register</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td></td>
<td>CONTEXTIDR, Context ID Register</td>
<td>Read/write only at PL1 or higher</td>
</tr>
<tr>
<td></td>
<td>TPIDRURW, User Read/Write Thread ID</td>
<td>Read/write at all privilege levels, including PL0</td>
</tr>
<tr>
<td></td>
<td>TPIDRURWO, User Read-only Thread ID</td>
<td>Read-only at PL0</td>
</tr>
<tr>
<td></td>
<td>TPIDRPW, PL1 only Thread ID</td>
<td>Read/write only at PL1 or higher</td>
</tr>
</tbody>
</table>

a. For accesses to 32-bit registers. More correctly, this is the primary coprocessor register.
b. Any attempt to execute an access that is not permitted results in an Undefined Instruction exception.
c. Some bits are common to the Secure and the Non-secure copies of the register, see SCtrl, System Control Register, on page B4-1707.
d. See ADFSR and AIFS, Auxiliary Data and Instruction Fault Status Registers, on page B4-1523. Register is IMPLEMENTATION DEFINED.
e. Banked only in an implementation that includes the FCSE. The FCSE PID Register is RAZ/WI if the FCSE is not implemented.

This is a (partial) list of the various control registers that you have to deal with. Nice to have it in one place.
B4 System Control Registers in a VMSA implementation
B4.1 VMSA System control registers descriptions, in register order

B4.1.130 SCTLR, System Control Register, VMSA

The SCTLR characteristics are:

Purpose
The SCTLR provides the top level control of the system, including its memory system.
This register is part of the Virtual memory control registers functional group.

Usage constraints
Only accessible from PL1 or higher.
Control bits in the SCTLR that are not applicable to a VMSA implementation read as the value that most closely reflects that implementation, and ignore writes.
In ARMv7, some bits in the register are read-only. These bits relate to non-configurable features of an ARMv7 implementation, and are provided for compatibility with previous versions of the architecture.

Configurations
In an implementation that includes the Security Extensions, the SCTLR:

• is Banked, with some bits common to the Secure and Non-secure copies of the register
• has write access to the Secure copy of the register disabled when the CP15DISABLE signal is asserted HIGH.

For more information, see Classification of system control registers on page B3-1451.

Attributes
A 32-bit RW register with an IMPLEMENTATION DEFINED reset value, see Reset value of the SCTL on page B4-1713. See also Reset behavior of CP14 and CP15 registers on page B3-1455.

Note
In an implementation that includes the Virtualization Extensions, some reset requirements apply to the Non-secure copy of SCTL.

In a VMSA v7 implementation, the SCTLR bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, UNK/ SBZP.</td>
</tr>
<tr>
<td>30</td>
<td>TE, bit[30] Thumb Exception enable. This bit controls whether exceptions are taken in ARM or Thumb state. The possible values of this bit are:</td>
</tr>
<tr>
<td>29</td>
<td>0 Exceptions, including reset, taken in ARM state.</td>
</tr>
<tr>
<td>28</td>
<td>1 Exceptions, including reset, taken in Thumb state.</td>
</tr>
<tr>
<td>27</td>
<td>In an implementation that includes the Security Extensions, this bit is Banked between the Secure and Non-secure copies of the register.</td>
</tr>
<tr>
<td>26</td>
<td>An implementation can include a configuration input signal that determines the reset value of the TE bit. If there is no configuration input signal to determine the reset value of this bit then it resets to 0 in an ARMv7-A implementation.</td>
</tr>
<tr>
<td>25</td>
<td>For more information about the use of this bit, see Instruction set state on exception entry on page B1-1182.</td>
</tr>
</tbody>
</table>

Table B3-45 on page B3-1493 shows the encodings of all of the registers in the Virtual memory control registers functional group.

This is the System Control Register, which has the all-important M bit in it, which turns on/off the MMU (i.e., virtual memory).
B4 System Control Registers in a VMSA implementation
B4.1 VMSA System control registers descriptions, in register order

B4.1.36 CONTEXTIDR, Context ID Register, VMSA

The CONTEXTIDR characteristics are:

- **Purpose**: CONTEXTIDR identifies the current Process Identifier (PROCID) and, when using the Short-descriptor translation table format, the Address Space Identifier (ASID).

- **Usage constraints**: Only accessible from PL1 or higher.

- **Configurations**: The register format depends on whether address translation is using the Long-descriptor or the Short-descriptor translation table format.

- **Attributes**: A 32-bit RW register with an UNKNOWN reset value. See also Reset behavior of CP14 and CP15 registers on page B3-1450.

Table B3-45 on page B3-1493 shows the encodings of all of the registers in the Virtual memory control registers functional group.

In a VMSA implementation, the CONTEXTIDR bit assignments are:

- **PROCID**, bits[31:0], when using the Long-descriptor translation table format
- **PROCID**, bits[31:8], when using the Short-descriptor translation table format
- **ASID**, bits[7:0], when using the Short-descriptor translation table format

**Using the CONTEXTIDR**

The value of the whole of this register is called the Context ID and is used by:

- the debug logic, for Linked and Unlinked Context ID matching, see Breakpoint debug events on page C3-2041 and Watchpoint debug events on page C3-2059
- the trace logic, to identify the current process.

The ASID field value is an identifier for a particular process. In the translation tables it identifies entries associated with a process, and distinguishes them from global entries. This means many cache and TLB maintenance operations take an ASID argument.

For information about the synchronization of changes to the CONTEXTIDR see Synchronization of changes to system control registers on page B3-1461. There are particular synchronization requirements when changing the ASID and Translation Table Base Registers, see Synchronization of changes of ASID and TTBR on page B3-1386.

---

When threads from multiple address spaces run, the hardware needs to be able to distinguish them. This is the register that does so. It tells the hardware “any PTE you load while running, attach this ASID to it when you put it into the TLB.” That way, when that process is swapped out and then is swapped back in later, it can still use its old mappings if they are still in the TLB.
B4.1.43 DACR, Domain Access Control Register, VMSA

The DACR characteristics are:

**Purpose**
DACR defines the access permission for each of the sixteen memory domains. This register is part of the Virtual memory control registers functional group.

**Usage constraints**
Only accessible from PL1 or higher.

**Configurations**
- If the implementation includes the Security Extensions, this register:
  - is Banked
  - has write access to the Secure copy of the register disabled when the CP15SDISABLE signal is asserted HIGH.
- In an implementation that includes the Large Physical Address Extension, this register has no function when TTBCR.EAE is set to 1, to select the Long-descriptor translation table format.

**Attributes**
A 32-bit RW register with an UNKNOWN reset value. For more information see Reset behavior of CP14 and CP15 registers on page B3-1450. Table B3-45 on page B3-1493 shows the encodings of all of the registers in the Virtual memory control registers functional group.

The DACR bit assignments are:

```
Dn, bits [(2n+1):2n]
```

Domain n access permission, where n = 0 to 15. Permitted values are:

- **0b00** No access. Any access to the domain generates a Domain fault.
- **0b01** Client. Accesses are checked against the permission bits in the translation tables.
- **0b10** Reserved, effect is UNPREDICTABLE.
- **0b11** Manager. Accesses are not checked against the permission bits in the translation tables.

For more information, see Domains, Short-descriptor format only on page B3-1362.

**Accessing the DACR**
To access the DACR, software reads or writes the CP15 registers with opc1 set to 0, crn set to c3, crm set to c0, and opc2 set to 0. For example:

- MRC p15, 0, <Rt>, c3, c0, 0 ; Read DACR into Rt
- MCR p15, 0, <Rt>, c3, c0, 0 ; Write Rt to DACR

This register is used to set up domains. I am not sure that we need to use them. My implementation does not.

**Build It, Load It, Run It**
Once you have it working, show us.